COMAL Kernal - 1985

Introduction

Superfluous or 1-productions have no semantics defined for them. Productions which just consist of a number of choices likewise have no semantics defined for them. We can assume that the semantics of A, where:

A ::= B | C | D

is the semantics of B, C, or D, and that we have a parser which can make the correct choice.

Conventions

::= "is defined as"
[ ] Optional
« » Metalinguistic Brackets
{ } May occur zero or more times
| Mutually exclusive alternatives

Note that the grammar is given mainly in iterative form, with the minimum of recursive definition.

Throughout this document "keywords" are shown in **UPPER CASE**.

Keywords are **RESERVED** and may **NOT** be used as identifiers. (A full list of reserved words is given at the end of this Kernal)

Note

This standard may be further revised. Prospective implementors of COMAL80 are advised to contact the group about any points not adequately covered by this document.

COMAL80 Standardisation Group,
Department of Computer & Information Science,
Linkoping University,
S-58183 Linkoping, Sweden
phone: +46-13-281000

Program Structure

«comal program» ::= «block»

«block» ::= {«declaration statement» |
«non declaration statement»}

The semantics of a block is the semantics of each of its component statements, taken in the context left over by executing the previous statements.

«declaration statement» ::=
«structured declaration statement» |
«unstructured declaration statement»

«non declaration statement» ::=
«structured statement» |
«unstructured statement»

«structured declaration statement» ::=
«procedure declaration» |
«function declaration»

«unstructured declaration statement» ::=
«dim statement» | «data statement»

«structured statement» ::=
«repetitive statement» |
«conditional statement»

The Structured Statements are composed of multiple lines. These lines are component lines. Each component line stands on its own as an input line and some syntax errors can be found. However further errors may be found when structures are checked, which can only be done when the program is complete.

«unstructured statement» ::=
«simple statement» «eol» |
«label statement» «eol» |
«eol»

«eol» ::= [«remark»] «newline»

more»

«remark» ::= //{«displayable character»}

A remark consists of a [possible empty] sequence of displayable characters, preceded by the sequence "//". The semantics of a remark is just its text with no interpretation by the system.

A comment can occur before any newline and will have no semantic effect on the preceding statements in the same line.

«newline» ::= implementation dependent

A newline is usually indicated by a carriage return, which is ASCII CHR$(13).

«displayable character» ::=
    implementation dependent

Displayable characters should include all displayable ASCII characters with values from 32 through 126.

«simple statement» ::= «return statement» |
    «stop statement» | «assignment statement» |
    «input statement» | «goto statement» |
    «restore statement» | «select statement» |
    «open statement» | «read statement» |
    «write statement» | «close statement» |
    «delete statement» | «print statement» |
    «zone statement» | «print using statement»|
    «procedure call statement» |
    «randomize statement»

Structured Statements

«repetitive statement» ::= «while statement» |
    «repeat statement» | «for statement»

«conditional statement» ::=
    «if statement» | «case statement»

«while statement» ::=
    WHILE «logical expression» DO «eol»
      «statement list»
    ENDWHILE «eol»

See the Extensions for the short one line while statement.

The expression is evaluated. If it is false (*zero*), control passes to the next statement after the ENDWHILE, and the only effect is any side effects generated by evaluating the expression. If the expression evaluates to true (*non-zero*), then the Statement list is executed. Control returns to the beginning of the WHILE statement, and the expression is re-evaluated. This pattern continues until the expression is found to be false. Control then passes to the statement following the ENDWHILE.

The statement list is indented when listed.

«statement list» ::=
    {«non declaration statement»}

The semantics of a sequence of non-declaration statements is the semantics of each component non-declaration statement in the context of the machine state which results from executing all the previous component non-declaration statements.

«repeat statement» ::=
    REPEAT «eol»
      «statement list»
    UNTIL «logical expression» «eol»

See the Extensions for the short one line repeat statement.

The statement list is executed. The expression is evaluated. If it is false (*zero*), control passes back to the first statement in the statement list. If the expression is true (*non-zero*), then control passes to the next statement following UNTIL.

more»

The statement list is indented when listed.

«for statement» ::= «short for statement» |
«long for statement»

«short for statement» ::=
FOR «for range» [«step»] DO
«simple statement» «eol»

«long for statement» ::=
FOR «for range» [«step»] DO «eol»
«statement list»
NEXT «control variable» «eol»

«for range» ::= «control variable» :=
«initial value» TO «final value»

«step» ::= STEP «step value»

«control variable» ::= «numeric identifier»

«initial value» ::= «numeric expression»

«final value» ::= «numeric expression»

«step value» ::= «numeric expression»

The For initialization is performed upon first entering the loop. This consists of evaluating the initial value and assigning the result to the control variable. Then the final value and the step value (if any) are evaluated. These three are evaluated once and for all and may not change during execution of the loop. The control variable is compared with the terminal value. In the case of a positive step value, if the control variable is less than or equal to the termination value, then the statement list is executed. In the case of a negative step value, it the control variable is greater then or equal to the termination value then the statement list is executed. In all other cases the statement list is not executed and control passes to the statement following the for statement. After each execution of the statement list, the value of the control variable is incremented by the step value. If the step part is null, then the control variable is incremented by one.

The statement list is indented when listed.

The semantics of the Short for statement is similar to the semantics of the long for statement, except that only a simple statement may occur after the DO.

If the control variable exceeds the final value on the first comparison, the statement list is not executed at all.

«if statement» ::=
«short if statement» | «long if statement»

«short if statement» ::=
IF «logical expression» THEN
«simple statement» «eol»

«long if statement» ::=
IF «logical expression» THEN «eol»
«statement list»
{ELIF «logical expression» THEN «eol»
«statement list»}
[ELSE «eol»
«statement list»]
ENDIF «eol»

«logical expression» ::= «numeric expression»

First the logical expression immediately following the IF is evaluated. If the value of the expression is true (non-zero), the statement list immediately following the THEN is executed, after which control passes to the statement following the ENDIF.

If the value of this expression is false (zero), and there are ELIF sections, the logical expressions of each of the ELIF statements are evaluated in order. As soon as one of these evaluates to true, the corresponding statement list is executed, and control then passes to the statement following the ENDIF.

more»

If none of the logical expressions *are* true, and there is an ELSE statement present, the statement list following the ELSE is executed, after which control passes to the statement after the ENDIF.

If none of the logical expressions *are* true, and there is no ELSE statement, control is passed to the statement after the ENDIF without any of the statement lists being executed.

The statement list is indented when listed.

The Short If statement behaves in the same way as the Long If statement, except that only a simple statement may occur after the THEN.

```
«case statement» ::=
    CASE «case selector» OF «eol»
    WHEN «choice list» «eol»
      «statement list»
    {WHEN «choice list» «eol»
      «statement list»}
    [OTHERWISE «eol»
      «statement list»]
    ENDCASE «eol»

«case selector» ::= «expression»

«choice list» ::= «numeric expression»
    {,«numeric expression»} |
    «string expression» {,«string expression»}
```

The case selector is evaluated. The choice lists are evaluated in the order in which they appear, until one is found with the same value as the case selector. When an identical value is found, then the statement list following the corresponding WHEN is executed. Control then passes to the statement following the ENDCASE. If none of the expressions in the WHEN choice lists match the CASE expression, the statement list following the OTHERWISE (if present) is executed. Control then passes to *the statement following the ENDCASE.*

Any Statements between the CASE .. OF line and first WHEN statement are ignored.

The statement list is indented when listed.

If none of the WHEN choice lists match the CASE selector, and no OTHERWISE clause exists, a runtime error occurs.

Structured Declarations

```
«procedure declaration» ::=
    PROC «procedure identifier»
    «head appendix» «eol»
      «procedure block»
    ENDPROC «procedure identifier» «eol»

«function declaration» ::=
    FUNC «function identifier»
    «head appendix» «eol»
      «function block»
    ENDFUNC «function identifier» «eol»

«function block» ::= «procedure block»

«return statement» ::= RETURN [«expression»]
```

Return causes the current procedure or function to be exited. A function **must** be exited through a return with an expression. A return with an expression is not allowed within a procedure. *Rather, a return without an expression may be used, but is not required.* A return of any sort may NOT be used within the main program.

```
«procedure block» ::=
    {«import statement»}
    {«unstructured declaration statement» |
    «non declaration statement»}

«head appendix» ::=
    [(«formal parameter list»)] [CLOSED]
```

more»

«procedure identifier» ::= «identifier»

«function identifier» ::=
    «numeric identifier» | «string identifier»

See the Extensions for the integer identifier.

«formal parameter list» ::=
    «formal parameter» {,«formal parameter»}

«formal parameter» ::=
    [REF] «variable identifier» |
    REF «variable identifier» «array indicator»

The Extensions also allow the REF to be optional for arrays.

«import statement» ::=
    IMPORT «import identifier»
    {,«import identifier»} «eol»

«import identifier» ::= «variable identifier» |
    «procedure identifier» | «function identifier»

«variable identifier» ::=
    «numeric identifier» | «string identifier»

«array indicator» ::= [({,})]

Functions and Procedures differ only in the way they are invoked and in the way they return values. A procedure is called by a procedure call, whereas a function is called by the use of the function identifier in an expression. The effect of a function invocation is to provide a single returned value of the same type as the function identifier (i.e. string or number). Procedures cannot return a value so procedure identifiers have no type (see also the RETURN statement).

The procedure or function can only be entered by calling it, and can only be exited by executing a RETURN statement or by control reaching the end of the procedure. The procedure or function identifier in the ENDPROC or ENDFUNC must match that in the heading. Parameters, other than arrays, are passed by value unless REF (pass by reference) is specified. Arrays are always passed by reference and REF must be included for them (*the Extensions allow REF to be optional for arrays too*). Their number of dimensions are indicated by commas. *No comma means one dimension, one comma means two dimensions, two commas mean three dimensions, etc.*

Procedures or functions are **Open** by default. **Closed** procedures or functions have their non-parameter variables allocated upon entry and these are lost when the procedure or function is exited.

The procedure block and function block are indented when listed.

Procedures or functions can be called recursively.

Unstructured Declarations

«dim statement» ::=
    DIM «declaration» {,«declaration»} «eol»

«declaration» ::=
    «numeric declaration» | «string declaration»

«numeric declaration» ::=
    «numeric identifier» («dimension part»)

«string declaration» ::=
    «string identifier» [(«dimension part»)]
    OF «length»

«dimension part» ::= «range» {,«range»}

«range» ::= [«lower bound»:] «upper bound»

«lower bound» ::= «numeric expression»

«upper bound» ::= «numeric expression»

more»

«length» ::= «numeric expression»

There can be arbitrary numbers of dimensions. If no lower bound is specified a default of 1 is assumed. If the upper bound is less than the lower bound, an error message will be issued. Each element of the items declared in the DIM statement is initialized. Numeric items are set to 0, and string items to the empty string. If an attempt is made to redimension an existing array a runtime error occurs. This means that one cannot make an array bigger by redimensioning it.

The Standards Group believes that dynamic string handling is preferable. If dynamic strings are implemented, the "OF «length»" part is optional. If the OF is present it sets the maximum length for the string. *The Extensions also allow the COMAL system to dimension a string to 40 characters maximum if the user does not specify otherwise.*

«data statement» ::=
    DATA «value» {,«value»} «eol»

All the DATA lists in the entire workspace are treated as one sequential "file" and are "consumed" by any READ statements that do NOT contain a file *designator.*

The Extensions allow data within CLOSED procedures and functions to be treated as local, not global to the whole program.

The execution of a RESTORE statement, without a label, causes the next READ operation to commence consuming input from the first DATA statement. If the RESTORE contains a label then the next READ is from the DATA statement following that label.

For good programming practice the data statement should only appear immediately before the end of the program or the end of a procedure.

«value» ::= [«sign»] «integer» |
    [«sign»] «real number» |
    «string constant» | TRUE | FALSE

«sign» ::= + | -

Because there is no boolean type the values TRUE and FALSE are in all respects equivalent to one and zero respectively.

False is always zero. True returns one, but when doing a comparison, any non-zero value is considered True.

Expressions

«expression» ::=
    «numeric expression» | «string expression»

«numeric expression» ::=
    [«numeric expression» OR] «logical term»

«logical term» ::=
    [«logical term» AND] «logical factor»

«logical factor» ::= [NOT] «relation»

«relation» ::=
    «string relation» | «arithmetic relation»

«string relation» ::= «string expression»
    «relational string operator»
    «string expression»

«relational string operator» ::=
    IN | «relational operator»

Syntax : «string1» IN «string2»

The two string expressions are evaluated. String2 is searched from the left until a copy of string1 is found. If no such copy is found the function returns FALSE (zero). Otherwise it returns the starting position of the first occurrence of string1. This must be non-zero and is therefore TRUE. If string1 is evaluated

more»

to the NULL string, then the value returned is 1. *NOTE: this is not how IN is implemented in some COMAL implementations, and is still under discussion. A proposal is currently submitted that specifies that the returned value should be false (zero).*

«arithmetic relation» ::=
    «formula» [«relational operator» «formula»]

«relational operator» ::=
    < | <= | = | >= | > | <>

«formula» ::= [«sign»] «arithmetic expression»

«arithmetic expression» ::=
    [«arithmetic expression» «adding operator»]
    «term»

«adding operator» ::= + | -

«term» ::=
    [«term» «multiplying operator»] «factor»

«multiplying operator» ::= * | / | DIV | MOD

The four *multiplying* operators have semantics as follows:

* and / perform the normal multiply and divide operations.

The DIV function is defined to take two arguments x and y and to return the next lowest integer less than or equal to the result of dividing x by y.

The MOD function also takes two arguments x and y (y must be positive) and returns (x - (x DIV y) * y).

These definitions of MOD and DIV are such as to give the following results:

```
   36 MOD 5 is 1      36 MOD (-5) is undefined
(-36) MOD 5 is 4   (-36) MOD (-5) is undefined
   35 MOD 5 is 0      35 MOD (-5) is undefined
(-35) MOD 5 is 0   (-35) MOD (-5) is undefined
   36 DIV 5 is 7      36 DIV (-5) is -8
(-36) DIV 5 is -8  (-36) DIV (-5) is 7
   35 DIV 5 is 7      35 DIV (-5) is -7
(-35) DIV 5 is -7  (-35) DIV (-5) is 7
```

Note: not all COMALs give these results for negative numbers.

«factor» ::= «operand» [^«factor»]

«operand» ::= («numeric expression») |
    «constant» | «numeric variable» |
    «numeric function call»

«constant» ::= «integer» | «real number» |
    TRUE | FALSE | *PI*

«real number» ::= «decimal number» [«exponent»]

«decimal number» ::= «integer»[.[«integer»]] |
    .«integer»

«exponent» ::= E [«sign»] «integer»

«integer» ::= «digit»{digit}

The allowable range of real and integer values is implementation dependent.

If an evaluation of a real expression results in underflow the value is set to zero. If the evaluation results in overflow, a run-time error occurs.

«numeric variable» ::=
    «numeric identifier» [(«subscript list»)]

«numeric identifier» ::= «real identifier»

«real identifier» ::= «identifier»

more»

«subscript list» ::= «subscript» {,«subscript»}
«subscript» ::= «numeric expression»

Rounding: If a real value is used as a subscript it is first rounded to the nearest integer value using the implicit rounding function which is defined to be (*the same as for the round statement in the Extensions*):

implicit_rounding_function(x) == int(x+0.5)

«numeric function call» ::=
    «numeric identifier»
    [(«actual parameter list»)]

«string expression» ::=
    «string operand» {+ «string operand»}

«string operand» ::= «string constant» |
    «string variable» | «string function call»

«string constant» ::= "{«displayable character»}"

«string variable» ::= «string identifier»
    [(«subscript list»)] [(«substring specifier»)]

«string identifier» ::= «identifier»$

«substring specifier» ::= «from»:«to»

The Extensions allow: [«from»:]«to»
If the from is omitted, only the character specified by the to is used.

«from» ::= «numeric expression»

«to» ::= «numeric expression»

«string function call» ::=
    «string identifier» [(«actual parameter list»)]
    [(«substring specifier»)]

«stop statement» ::= STOP

The Extensions allow a message after the STOP.

The Program is suspended and control returns to the COMAL system. All variables retain their current values, and the program may later be restarted at the statement immediately following the STOP.

«assignment statement» ::=
    «assignment» {;«assignment»}

«assignment» ::= «numeric assignment» |
    «string assignment»

«numeric assignment» ::=
    «numeric variable» := «numeric expression»

«string assignment» ::=
    «string variable» := «string expression»

In an expression containing both real and integer values the integers are "promoted" to reals and the expression evaluates to a real.

Precedence of Operators:

The precedence of the various operators in COMAL is shown in the following table. 1 is the highest precedence. Operators of the same precedence are evaluated left to right, except for exponentiation which is evaluated from right to left.

1) Exponentiation	^
2) Multiplying Operators	* | / | DIV | MOD
3) Adding Operators (Including Unary Minus)	+ | -
4) Relational Operators	= | <> | < | > | <= | >= | IN
5) Logical Negation	NOT
6) Logical ANDing	AND
7) Logical ORing	OR

If an operand is an expression within parentheses, the value to the operand is the value of that bracketed expression. It follows that the order of evaluation can be modified by inserting matching pairs of parentheses.

more»

If the string variable is given without a substring specifier then:

1. if the length of the string expression is greater than the length of the string variable, the string expression is truncated to the right.

2. if the string expression is shorter than or equal to the length of the string variable, the value is assigned and the actual length of the sting variable is set to the length of the string expression.

If the string variable refers to a substring then:

1. if the length of the string expression is greater than the length of the substring, the string expression is truncated to the right. If the string expression is shorter than or equal to the length of the substring, the value is assigned and the remaining part of the substring is space filled.

2. if the start index for the substring is greater than the actual length of the string variable + 1, a run-time error occurs.

Input Statement

«input statement» ::=
    INPUT [«string constant»:] «variable»
    «print end» |
    INPUT «file designator»: «variable list»

The Extensions also allow the string constant to be a string expression.

«variable list» ::= «variable» {,«variable»}

«variable» ::= «numeric variable» |
    «string variable»

«file designator» ::=
    FILE «channel number» [,«record number»]

«channel number» ::= «numeric expression»

«record number» ::= «numeric expression»

The «string constant» is used as a prompt. If the prompt is absent then a system standard prompt is supplied. The standard prompt is implementation dependent. (*We would like the standard default prompt to be declared. We suggest ? as the prompt.*) The system waits for the user to input a value. If the user makes a mistake and types in a value that does not match the type of the variable, the system prints an error message and re-issues the prompt. If «print end» is not specified the following print position will be the first position on the next line. If the «print end» is specified, the rules from the print statement apply.

If the file designator is present then no prompt is allowed as the input is read from an ASCII file. A file which is read by means of an INPUT statement is ASCII, whereas a file read by means of READ is binary.

Goto Statement

«goto statement» ::= GOTO «label identifier»

The GOTO statement causes transfer of control to the label statement with the corresponding label identifier. The GOTO is restricted in where it can jump to.

1. A GOTO cannot transfer control into a structured statement. One can have a label statement within a structured statement, but only statements inside the structured statement may GOTO it.

2. A GOTO cannot transfer control into or out of a procedure.

more»

3. Subject to 1. and 2. above, a GOTO may transfer control out of a structured statement.

«restore statement» ::=
    RESTORE [«label identifier»]

The pointer to the next data item is reset to point to the first item in the (textually) first DATA statement. If the label is included the pointer is restored only to the DATA statement immediately following that label.

«label statement» ::= «label identifier»:

«label identifier» ::= «identifier»

The Label statement is included as an unstructured statement. It may appear within a structured statement, although it only has scope (i.e. can only be jumped to) within the structured statement. Note that one may not include a label statement on the same line as another statement.

A label statement is also used to separate data statements (see the RESTORE statement).

File Handling

«select statement» ::=
    SELECT «type» «device specifier»

[,«dev info»] had been included after «device specifier». It was used in an older version of the Kernal, but since then «dev info» was deleted from the Kernal and all references to it also should be deleted.

«type» ::= OUTPUT

The Extensions should also allow INPUT as a type.

«device specifier» ::= «string expression»

This string expression can be a file name or peripheral device identification (such as a printer).

«open statement» ::=
    OPEN FILE «channel number»,«file name»,
    «mode»

«file name» ::= «string expression»

The file name is assumed to exist on the default disk drive, unless another drive or device is specified. To specify a specific drive, precede the actual file name with [«drive id»:] (the colon separates the id from the file name). Computer systems use various methods of identifying a disk drive. However, the COMAL system can accept one method, and then (transparent to the user) convert it into the method required by the computer system. This has not been decided yet.

«mode» ::= READ | WRITE | APPEND |
    RANDOM «record length» [«random mode»]

«random mode» ::= READONLY | WRITEONLY

«record length» ::= «numeric expression»

If mode is random all records have the specified length. The data, if any, in the record need not fill the record. The string expression should evaluate to a file name. If the file does or does not exist, the system will respond according to the following plan:

mode	yes	no
read	open	run-time error
write	run-time error	create
append	open	create
random	open	create
random readonly	open	run-time error
random writeonly	open	create

If no error has occurred, the file will be associated with the channel number given.

more»

«read statement» ::= READ «variable list» |
    READ «file designator»: «variable list»

If the variable type in the read statement is incompatible with the type of the value in the data statement, a run-time error occurs.

«write statement» ::=
    WRITE «file designator»: «variable list»

A record number in the file designator is valid only if the file is opened in random access mode. Otherwise the presence of a record number causes an error to occur. The expressions are written out in binary (if they are numeric) or in ASCII (if they are text) to the file opened on the given channel. If no file is attached to the channel, an error results. If the file is random access, then the combined bulk of all the data in the expression list should be less than or equal to the record length.

«delete statement» ::=
    DELETE «file name»

If the named file is currently open, a runtime error occurs. If it is closed, it is deleted, and if it is reopened, it will be a new file. It is permitted to delete a non-existing file, and no error results.

«close statement» ::=
    CLOSE [FILE «channel number»]

The file attached to the channel specified by channel number is disconnected. If the channel number given is not attached to any file, an error occurs. If no channel number is specified than ALL currently open files are closed. *If there are no open files, no error results.*

Print Statement

«print statement» ::= PRINT «output list» |
    PRINT «file designator»: «output list»

«output list» ::= [«print list» [«print end»]]

«print list» ::= «print element»
    {«print separator» «print element»}

«print element» ::= «expression» | «tab function»

«print end» ::= «print separator»

«print separator» ::= , | ;

We also are suggesting the ! or : as a null separator for no spaces between items.

«tab function» ::= TAB(«numeric expression»)

TAB causes the next printed item to go in the column position given by the expression. The first position allowed is defined to be 1. A TAB with a non-positive argument gives a runtime error. The maximum argument permitted is implementation dependent. If the argument is less than the current cursor position, it has no effect.

If a file designator is specified, then the ASCII output goes out to the file rather than to the user's terminal. Formatted information may be written out to ASCII files.

Normally the expressions are printed out using a system-defined format for the particular type of expression.

The print line is divided into print zones. The zone width is determined by the zone statement. The default zone width is 0. An odd sized zone may occur at the end of the print line. Printing zone divided lines may be compared to using the tabulator key on an ordinary typewriter. This means that after a print element is output the tabulator key is pressed. As a consequence of this the next print element starts in the leftmost position of the next zone, and that printing a print element of the same length as the zone width

more»

leaves the following zone space filled. Zone width 0 means that no spacing between print elements occurs.

A semicolon used as print separator is output as one space. *We propose the exclamation point or colon as a null print separator, always outputing nothing between items.* The effect of a «print end» is the same as of a «print separator». If no «print end» is specified, printing is continued in the first position on the next line. A print statement with no «print list» causes output of a newline.

«print using statement» ::=
    PRINT USING «format info»: «using list» [«print end»] |
    PRINT «file designator»: USING «format info»: «using list» [«print end»]

«using list» ::=
    «using element» {,«using element»}

«using element» ::= «numeric expression»

«format info» ::= «string expression»

In the PRINT USING statement the format is determined from the string in the format clause. The meaning of the characters in the string of the Format Info is as follows: A substring of one or more embedded hash signs (#), optionally containing a single point (.), will be substituted on output by the value of a corresponding numeric expression. The expression will be printed out with as many digits precision as specified by hash signs to the right of the point. If no point is specified, neither the decimal point nor any fractional part is printed. Leading zeroes in the integer part are replaced by *spaces*; trailing zeroes in the fractional part are printed. If the expression has an integer part which is too large to be represented in the field, the field is printed out as all *asterisks* "*".

Procedure Call

«procedure call statement» ::=
    [EXEC] «procedure identifier» [(«actual parameter list»)]

Syntactically, the only restriction on actual parameters is that they must be expressions. However, in order to send results back via a parameter passed by reference, the actual parameter must have a l-value. (e.g. the l-value of X is the address X, but the expression X+Y has no l-value. *A REF parameter can only pass results to a variable, not a constant or expression.*)

A procedure may have side effects, both **through parameters** passed by reference and through shared variables.

SCOPE : Within a given scope, the first use of a name, whether as a procedure name, function name, label or variable name (whether DIMmed or not) defines the names type. Any subsequent attempt to redefine this name is an error.

Some systems do allow a name to be used in more than one way: test$ and test#.

BINDING :
1. Procedure and function names must be IMPORTed into CLOSED procedures.
2. OPEN procedures or functions canNOT be IMPORTed. *Some systems do allow OPEN procedures to be IMPORTed.*

Advisory Note

To ensure portability of software, users procedures should NOT rely on their formal parameters being available to called procedures.

«actual parameter list» ::=
    «actual parameter» {,«actual parameter»}

«actual parameter» ::= «expression»

more»

«zone statement» ::= ZONE «numeric expression»

ZONE sets the default spacing for items output with the PRINT statement.

The Extensions allow ZONE to be used as a function that returns the current zone setting.

«randomize statement» ::=
RANDOMIZE [«numeric expression»]

Prior to the execution of any program the random number generator is seeded by a random value.

The execution of a randomize statement without a numeric expression means that the random number generator is seeded by a random value.

The execution of a randomize statement with a numeric expression means that the random number generator is seeded by the value of the numeric expression.

«identifier» ::= «letter» {«letter» | «digit» | _}

The apostrophe ' may also be used as part of an identifier, just like the underscore.

If underscore is not available on the terminal to be used, another character may be chosen.

«letter» ::= implementation dependent

A letter must include all upper and lower case ASCII letters. It may optionally include other national characters.

«digit» ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Standard Built-in Functions

COS(«numeric expression»)
SIN(«numeric expression»)
TAN(«numeric expression»)
ATN(«numeric expression»)

-- All the trigonometric function take a numeric argument whose value is assumed to be in radians. The value returned is real. An invalid argument causes a runtime error.

Some systems allow the user to choose to use degrees rather than radians.

ABS(«numeric expression»)

-- Returns the **absolute** value of the expression.

LOG(«numeric expression»)
EXP(«numeric expression»)

-- The LOG and EXP functions find the natural log and exponent values of a REAL argument.

SQR(«numeric expression»)

-- *SQR requires that the numeric expression be positive. Non-positive values result in a run time error.*

INT(«numeric expression»)

-- returns the next smallest integer value. Thus INT(-3.5) returns -4.

SGN(«numeric expression»)

-- returns -1 for a negative number, 0 for 0 and +1 for a positive number.

RND

-- returns a "random" value greater than or equal to zero, and less than one.

RND(«numeric expression»,«numeric expression»)

-- returns a "random" *integer* in the range given (*inclusive*).

more»

LEN(«string expression»)

-- returns the *current* length of a string

ORD(«string expression»)

-- returns the internal representation (*ASCII*) of the first character of its argument. *The null string results in a value of -1 (proposed).*

VAL(«string expression»)

-- returns the numeric equivalent of its string expression argument. The argument can have the format:

[«sign»]«integer»[.«integer»][E«integer»]

or

[«sign»].«integer»[E«integer»]

No blanks are permitted in the string.

STR$(«numeric expression»)

-- PRINT x and PRINT STR$(x) always produce the same result.

CHR$(«numeric expression»)

-- CHR$ takes an internal representation (*ASCII*) and returns the equivalent character. It is the inverse of ORD.

EOF(«numeric expression»)

-- the numeric expression gives a channel number. EOF becomes true when there are no more records to be read. EOF of a closed file causes a runtime error. EOF becomes true if an empty file is opened.

EOD

-- EOD becomes true in the same way *as EOF* except that the file is the internal data file.

STANDARD EXTENSIONS

Integer Type Variables

To allow variables of type integer, replace the production:

«numeric identifier» ::= «real identifier»

with the two productions:

«numeric identifier» ::= «real identifier» |
«integer identifier»

«integer identifier» ::= «identifier»#

Some flexibility between real and integers is allowed. If an integer expression is assigned to a real variable an implicit FLOAT() takes place without any errors. If a real expression is assigned to an integer variable, an implicit FIX() takes place. Two functions for rounding are provided. These are ROUND() and ROUNDEVEN().

Short WHILE and REPEAT

One line versions of these statements are provided by modifying the productions for the Repetitive Statements.

«while statement» ::= «short while statement» |
«long while statement»

«short while statement» ::= WHILE
«logical expression» DO «simple statement»
«eol»

«long while statement» ::=
WHILE «logical expression» DO «eol»
«statement list»
ENDWHILE «eol»

«repeat statement» ::= «short repeat statement»|
«long repeat statement»

more»

«short repeat statement» ::= REPEAT
«simple statement» UNTIL
«logical expression» «eol»

«long repeat statement» ::=
REPEAT *«eol»*
«statement list»
UNTIL «logical expression» «eol»

STOP With a Message

Instead of printing line numbers a STOP statement can result in a message being output. The syntax is:

«stop statement» ::= STOP[«string expression»]

Static Strings

If dynamic strings are NOT implemented a default string-length of 40 (forty) is given to an unDIMmed string variable when it is first encountered.

Parameter Lists to Procedures

To drop the requirement that the parameter list to a procedure be parenthesised, replace the production

«procedure call statement» ::=
[EXEC] «procedure identifier»
[(«actual parameter list»)]

with the following productions.

«procedure call statement» ::= [EXEC]
«procedure identifier»[(«parameter list»)]
«parameter list» ::=
«parenthesised parameter list» |
«actual parameter list»

«parenthesised parameter list» ::=
(«actual parameter list»)

Add to the built in functions:

«zone function» ::= ZONE

This returns the current zone setting.

«rounding function» ::=
ROUND(«numeric expression»)

This rounds a value to the nearest integer.

Add to the simple statement:

«clear screen statement» ::= PAGE

Reserved Words in COMAL 80

ABS	FLOAT	REF
AND	FOR	REPEAT
APPEND	FUNC	RESTORE
ATN	GOTO	RETURN
CASE	IF	RND
CHR$	IMPORT	ROUND
COS	IN	ROUNDEVEN
CLOSE	INPUT	SGN
CLOSED	INT	SIN
DATA	LEN	SQR
DIM	LOG	STEP
DIV	MOD	STOP
DO	NEXT	STR$
ELIF	NOT	TAB
ELSE	OF	TAN
ENDCASE	ON	THEN
ENDFUNC	OPEN	TO
ENDPROC	OR	TRUE
ENDWHILE	ORD	UNTIL
EOD	OTHERWISE	USING
EOF	PRINT	VAL
EXEC	PROC	WHEN
EXP	RANDOM	WHILE
FALSE	RANDOMIZE	WRITE
FILE	READ	WRITEONLY
FIX	READONLY	*PAGE*

[This is the April 1985 COMAL Kernal, the latest edition that we have. Items in italics are our own clarifications and comments, which we are proposing to be added into the Kernal.] ■